08｜入口函数与包初始化：搞清Go程序的执行次序

2021-10-29 Tony Bai

《Tony Bai · Go语言第一课》 课程介绍 >

讲述：Tony Bai
时长 17:34 大小 16.09M

你好，我是 Tony Bai。

在刚开始学习 Go 语言的时候，我们可能经常会遇到这样一个问题：一个 Go 项目中有数十个 Go 包，每个包中又有若干常量、变量、各种函数和方法，那 Go 代码究竟是从哪里开始执行的呢？后续的执行顺序又是什么样的呢？

事实上，了解这门语言编写应用的执行次序，对我们写出结构合理、逻辑清晰的程序大有裨益，无论你用的是归属为哪种编程范式（Paradigm）的编程语言，过程式的、面向对象的、函数式的，或是其他编程范式的，我都建议你深入了解一下。

所以今天这节课，我就带你来了解一下 Go 程序的执行次序，这样在后续阅读和理解 Go 代码的时候，你就好比拥有了“通往宝藏的地图”，可以直接沿着 Go 代码执行次序这张“地图”去阅读和理解 Go 代码了，不会在庞大的代码库中迷失了。

Go 程序由一系列 Go 包组成，代码的执行也是在各个包之间跳转。和其他语言一样，Go 也拥有自己的用户层入口：**main 函数**。这节课我们就从 main 函数入手，逐步展开，最终带你掌握 Go 程序的执行次序。

那么下面，我们就先来看看 Go 应用的入口函数。

main.main 函数：Go 应用的入口函数

Go 语言中有一个特殊的函数：main 包中的 main 函数，也就是 **main.main**，它是所有 Go 可执行程序的用户层执行逻辑的入口函数。Go 程序在用户层面的执行逻辑，会在这个函数内按照它的调用顺序展开。

main 函数的函数原型是这样的：

复制代码

```
package main

func main() {
    // 用户层执行逻辑
    ... ...
}
```

你会发现，main 函数的函数原型非常简单，没有参数也没有返回值。而且，Go 语言要求：**可执行程序的 main 包必须定义 main 函数，否则 Go 编译器会报错。**在启动了多个 Goroutine（Go 语言的轻量级用户线程，后面我们会详细讲解）的 Go 应用中，main.main 函数将在 Go 应用的主 Goroutine 中执行。

不过很有意思的是，在多 Goroutine 的 Go 应用中，相较于 main.main 作为 Go 应用的入口，main.main 函数返回的意义其实更大，因为 main 函数返回就意味着整个 Go 程序的终结，而且你也不用管这个时候是否还有其他子 Goroutine 正在执行。

另外还值得我们注意的是，除了 main 包外，其他包也可以拥有自己的名为 main 的函数或方法。但按照 Go 的可见性规则（小写字母卡头的标识符为非导出标识符），非 main 包中自定义的 main 函数仅限于包内使用，就像下面代码这样，这是一段在非 main 包中定义 main 函数的代码片段：

复制代码

```
package pkg1

import "fmt"

func Main() {
    main()
}

func main() {
    fmt.Println("main func for pkg1")
}
```

你可以看到，这里 main 函数就主要是用来在包 pkg1 内部使用的，它是没法在包外使用的。

好，现在我们已经了解了 Go 应用的入口函数 main.main 的特性。**不过对于 main 包的 main 函数来说，你还需要明确一点，就是它虽然是用户层逻辑的入口函数，但它却不一定是用户层第一个被执行的函数。**

这是为什么呢？这跟 Go 语言的另一个函数 init 有关。

init 函数：Go 包的初始化函数

除了前面讲过的 main.main 函数之外，Go 语言还有一个特殊函数，它就是用于**进行包初始化的 init 函数**了。

和 main.main 函数一样，init 函数也是一个无参数无返回值的函数：

复制代码

```
func init() {
    // 包初始化逻辑
    ... ...
}
```

那我们现在回到前面这个“main 函数不一定是用户层第一个被执行的函数”的问题，其实就是因为，如果 main 包依赖的包中定义了 init 函数，或者是 main 包自身定义了 init 函数，那么 Go 程序在这个包初始化的时候，就会自动调用它的 init 函数，因此这些 init 函数的执行就都会发生在 main 函数之前。

不过对于 init 函数来说，我们还需要注意一点，就是在 Go 程序中我们不能手工显式地调用 init，否则就会收到编译错误，就像下面这个示例，它表示的手工显式调用 init 函数的错误做法：

复制代码

```
package main

import "fmt"

func init() {
  fmt.Println("init invoked")
}

func main() {
   init()
}
```

这样，在构建并运行上面这些示例代码之后，Go 编译器会报下面这个错误：

复制代码

```
$go run call_init.go
./call_init.go:10:2: undefined: init
```

实际上，Go 包可以拥有不止一个 init 函数，每个组成 Go 包的 Go 源文件中，也可以定义多个 init 函数。

所以说，在初始化 Go 包时，Go 会按照一定的次序，逐一、顺序地调用这个包的 init 函数。一般来说，先传递给 Go 编译器的源文件中的 init 函数，会先被执行；而同一个源文件中的多个 init 函数，会按声明顺序依次执行。

那么，现在我们就知晓了 main.main 函数可能并不是第一个被执行的函数的原因了。所以，当我们要在 main.main 函数执行之前，执行一些函数或语句的时候，我们只需要将它放入 init 函数中就可以了。

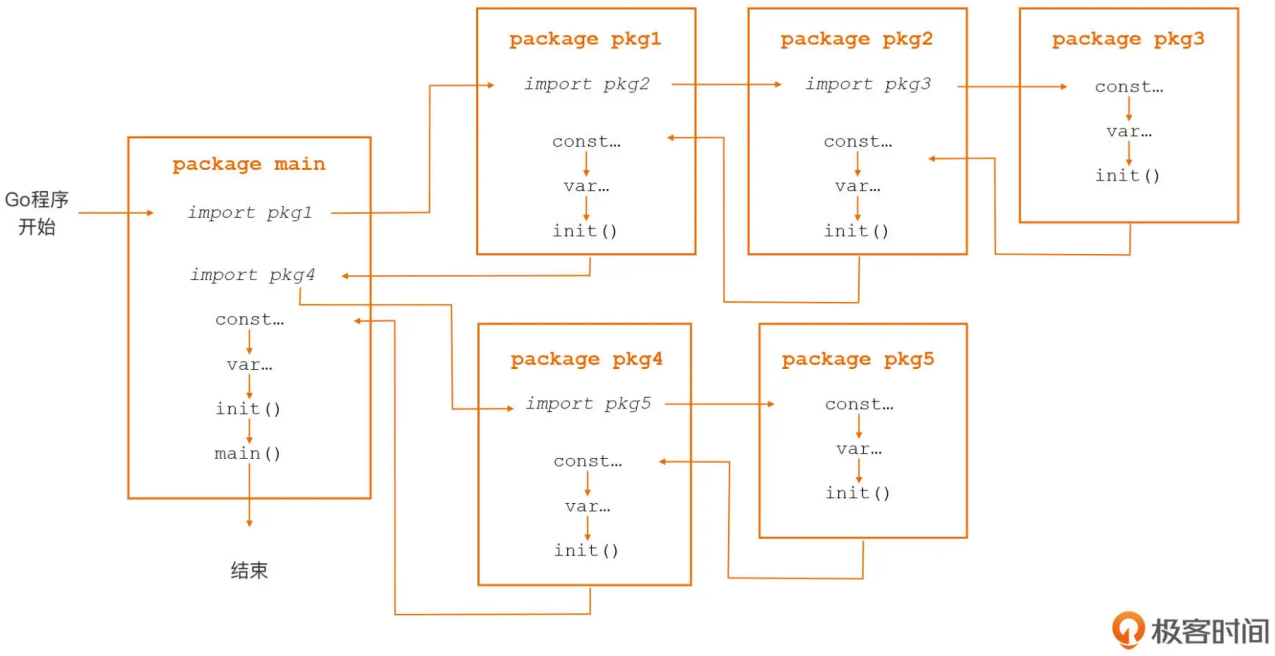
了解了这两个函数的执行顺序之后，我们现在就来整体地看看，一个 Go 包的初始化是以何种次序和逻辑进行的。

Go 包的初始化次序

我们从程序逻辑结构角度来看，Go 包是程序逻辑封装的基本单元，每个包都可以理解为是一个“自治”的、封装良好的、对外部暴露有限接口的基本单元。一个 Go 程序就是由一组包组成的，程序的初始化就是这些包的初始化。每个 Go 包还会有自己的依赖包、常量、变量、init 函数（其中 main 包有 main 函数）等。

在这里你要注意：**我们在阅读和理解代码的时候，需要知道这些元素在在程序初始化过程中的初始化顺序，这样便于我们确定在某一行代码处这些元素的当前状态。**

下面，我们就通过一张流程图，来了解学习下 Go 包的初始化次序：



Go包的初始化次序

这里，我们来看看具体的初始化步骤。

首先，main 包依赖 pkg1 和 pkg4 两个包，所以第一步，Go 会根据包导入的顺序，先去初始化 main 包的第一个依赖包 pkg1。

第二步，Go 在进行包初始化的过程中，会采用“深度优先”的原则，递归初始化各个包的依赖包。在上图里，pkg1 包依赖 pkg2 包，pkg2 包依赖 pkg3 包，pkg3 没有依赖包，于是 Go 在 pkg3 包中按照“常量 -> 变量 -> init 函数”的顺序先对 pkg3 包进行初始化；

紧接着，在 pkg3 包初始化完毕后，Go 会回到 pkg2 包并对 pkg2 包进行初始化，接下来再回到 pkg1 包并对 pkg1 包进行初始化。在调用完 pkg1 包的 init 函数后，Go 就完成了 main 包的第一个依赖包 pkg1 的初始化。

接下来，Go 会初始化 main 包的第二个依赖包 pkg4，pkg4 包的初始化过程与 pkg1 包类似，也是先初始化它的依赖包 pkg5，然后再初始化自身；

然后，当 Go 初始化完 pkg4 包后也就完成了对 main 包所有依赖包的初始化，接下来初始化 main 包自身。

最后，在 main 包中，Go 同样会按照“常量 -> 变量 -> init 函数”的顺序进行初始化，执行完这些初始化工作后才正式进入程序的入口函数 main 函数。

现在，我们可以通过一段代码示例来验证一下 Go 程序启动后，Go 包的初始化次序是否是正确的，示例程序的结构如下：

复制代码

```
prog-init-order
├── go.mod
├── main.go
├── pkg1
│   └── pkg1.go
├── pkg2
│   └── pkg2.go
└── pkg3
    └── pkg3.go
```

我们设定的各个包的依赖关系如下：

main 包依赖 pkg1 包和 pkg2 包；

pkg1 包和 pkg2 包都依赖 pkg3 包。

这里我只列出了 main 包的代码，pkg1、pkg2 和 pkg3 包的代码与 main 包都是类似的，你可以自己尝试去列一下。

复制代码

```
package main

import (
    "fmt"

    _ "github.com/bigwhite/prog-init-order/pkg1"
    _ "github.com/bigwhite/prog-init-order/pkg2"
)

var (
    _  = constInitCheck()
    v1 = variableInit("v1")
    v2 = variableInit("v2")
)

const (
    c1 = "c1"
    c2 = "c2"
)

func constInitCheck() string {
    if c1 != "" {
        fmt.Println("main: const c1 has been initialized")
    }
    if c1 != "" {
        fmt.Println("main: const c2 has been initialized")
    }
    return ""
}

func variableInit(name string) string {
    fmt.Printf("main: var %s has been initialized\n", name)
    return name
}

func init() {
    fmt.Println("main: first init func invoked")
}

func init() {
    fmt.Println("main: second init func invoked")
```

```
}

func main() {
    // do nothing
}
```

我们可以看到，在 main 包中其实并没有使用 pkg1 和 pkg2 中的函数或方法，而是直接通过空导入的方式“触发”pkg1 包和 pkg2 包的初始化（pkg2 包也是通过空导入的方式依赖 pkg3 包的），下面是这个程序的运行结果：

复制代码

```
$go run main.go
pkg3: const c has been initialized
pkg3: var v has been initialized
pkg3: init func invoked
pkg1: const c has been initialized
pkg1: var v has been initialized
pkg1: init func invoked
pkg2: const c has been initialized
pkg2: var v has been initialized
pkg2: init func invoked
main: const c1 has been initialized
main: const c2 has been initialized
main: var v1 has been initialized
main: var v2 has been initialized
main: first init func invoked
main: second init func invoked
```

你看，正如我们预期的那样，Go 运行时是按照“pkg3 -> pkg1 -> pkg2 -> main”的顺序，来对 Go 程序的各个包进行初始化的，而在包内，则是以“常量 -> 变量 -> init 函数”的顺序进行初始化。此外，main 包的两个 init 函数，会按照在源文件 main.go 中的出现次序进行调用。

还有一点，pkg1 包和 pkg2 包都依赖 pkg3 包，但根据 Go 语言规范，一个被多个包依赖的包仅会初始化一次，因此这里的 pkg3 包仅会被初始化了一次。

所以简而言之，记住 Go 包的初始化次序并不难，你只需要记住这三点就可以了：

依赖包按“深度优先”的次序进行初始化；

每个包内按以“常量 -> 变量 -> init 函数”的顺序进行初始化；

包内的多个 init 函数按出现次序进行自动调用。

到这里，我们已经知道了 Go 程序中包的初始化次序，也了解了每个包中常量、变量以及 init 函数的运行次序，以及 init 函数作为包初始化函数的一些特性。

搞完了这些最主线的内容之后，不知你有没有发现，我们好像还忘记了一件事：我们好像忘记分析 init 函数的用途了？别急，我们现在就把这落下的功课补上，看看作为 Go 包初始化函数的 init 函数，在日常 Go 语言开发中怎么来使用呢？

init 函数的用途

其实，init 函数的这些常用用途，与 init 函数在 Go 包初始化过程中的次序密不可分。我们前面讲过，Go 包初始化时，init 函数的初始化次序在变量之后，这给了开发人员在 init 函数中对包级变量进行进一步检查与操作的机会。

这里我们先来看 init 函数的第一个常用用途：重置包级变量值。

init 函数就好比 Go 包真正投入使用之前唯一的“质检员”，负责对包内部以及暴露到外部的包级数据（主要是包级变量）的初始状态进行检查。在 Go 标准库中，我们能发现很多 init 函数被用于检查包级变量的初始状态的例子，标准库 flag 包对 init 函数的使用就是其中的一个，这里我们简单来分析一下。

flag 包定义了一个导出的包级变量 CommandLine，如果用户没有通过 flag.NewFlagSet 创建新的代表命令行标志集合的实例，那么 CommandLine 就会作为 flag 包各种导出函数背后，默认的代表命令行标志集合的实例。

而在 flag 包初始化的时候，由于 init 函数初始化次序在包级变量之后，因此包级变量 CommandLine 会在 init 函数之前被初始化了，你可以看一下下面的代码：

复制代码

```
var CommandLine = NewFlagSet(os.Args[0], ExitOnError)

func NewFlagSet(name string, errorHandling ErrorHandling) *FlagSet {
    f := &FlagSet{
        name:          name,

```

```
        errorHandling: errorHandling,
    }
    f.Usage = f.defaultUsage
    return f
}

func (f *FlagSet) defaultUsage() {
    if f.name == "" {
        fmt.Fprintf(f.Output(), "Usage:\n")
    } else {
        fmt.Fprintf(f.Output(), "Usage of %s:\n", f.name)
    }
    f.PrintDefaults()
}
```

我们可以看到，在通过 NewFlagSet 创建 CommandLine 变量绑定的 FlagSet 类型实例时，CommandLine 的 Usage 字段被赋值为 defaultUsage。

也就是说，如果保持现状，那么使用 flag 包默认 CommandLine 的用户就无法自定义 usage 的输出了。于是，flag 包在 init 函数中重置了 CommandLine 的 Usage 字段：

复制代码

```
func init() {
    CommandLine.Usage = commandLineUsage // 重置CommandLine的Usage字段
}

func commandLineUsage() {
    Usage()
}

var Usage = func() {
    fmt.Fprintf(CommandLine.Output(), "Usage of %s:\n", os.Args[0])
    PrintDefaults()
}
```

这个时候我们会发现，CommandLine 的 Usage 字段，设置为了一个 flag 包内的未导出函数 commandLineUsage，后者则直接使用了 flag 包的另外一个导出包变量 Usage。这样，就可以通过 init 函数，将 CommandLine 与包变量 Usage 关联在一起了。

然后，当用户将自定义的 usage 赋值给了 flag.Usage 后，就相当于改变了默认代表命令行标志集合的 CommandLine 变量的 Usage。这样当 flag 包完成初始化后，CommandLine 变量便处于一个合理可用的状态了。

init 函数的第二个常用用途，是实现对包级变量的复杂初始化。

有些包级变量需要一个比较复杂的初始化过程，有些时候，使用它的类型零值（每个 Go 类型都具有一个零值定义）或通过简单初始化表达式不能满足业务逻辑要求，而 init 函数则非常适合完成此项工作，标准库 http 包中就有这样一个典型示例：

复制代码

```
var (
    http2VerboseLogs    bool // 初始化时默认值为false
    http2logFrameWrites bool // 初始化时默认值为false
    http2logFrameReads  bool // 初始化时默认值为false
    http2inTests        bool // 初始化时默认值为false
)

func init() {
    e := os.Getenv("GODEBUG")
    if strings.Contains(e, "http2debug=1") {
        http2VerboseLogs = true // 在init中对http2VerboseLogs的值进行重置
    }
    if strings.Contains(e, "http2debug=2") {
        http2VerboseLogs = true // 在init中对http2VerboseLogs的值进行重置
        http2logFrameWrites = true // 在init中对http2logFrameWrites的值进行重置
        http2logFrameReads = true // 在init中对http2logFrameReads的值进行重置
    }
}
```

我们可以看到，标准库 http 包定义了一系列布尔类型的特性开关变量，它们默认处于关闭状态（即值为 false），但我们可以通过 GODEBUG 环境变量的值，开启相关特性开关。

可是这样一来，简单地将这些变量初始化为类型零值，就不能满足要求了，所以 http 包在 init 函数中，就根据环境变量 GODEBUG 的值，对这些包级开关变量进行了复杂的初始化，从而保证了这些开关变量在 http 包完成初始化后，可以处于合理状态。

说完了这个，我们现在来讲 init 函数的第三个常用用途：在 init 函数中实现“注册模式”。

为了让你更好地理解，首先我们来看一段使用 lib/pq 包访问 PostgreSQL 数据库的代码示例：

复制代码

```
import (
    "database/sql"
    _ "github.com/lib/pq"
)

func main() {
    db, err := sql.Open("postgres", "user=pqgotest dbname=pqgotest sslmode=ver
    if err != nil {
        log.Fatal(err)
    }

    age := 21
    rows, err := db.Query("SELECT name FROM users WHERE age = $1", age)
    ...
}
```

其实，这是一段“神奇”的代码，你可以看到示例代码是以空导入的方式导入 lib/pq 包的，main 函数中没有使用 pq 包的任何变量、函数或方法，这样就实现了对 PostgreSQL 数据库的访问。而这一切的奥秘，全在 pq 包的 init 函数中：

复制代码

```
func init() {
    sql.Register("postgres", &Driver{})
}
```

这个奥秘就在，我们其实是利用了用空导入的方式导入 lib/pq 包时产生的一个“副作用”，也就是 lib/pq 包作为 main 包的依赖包，它的 init 函数会在 pq 包初始化的时候得以执行。

从上面代码中，我们可以看到在 pq 包的 init 函数中，pq 包将自己实现的 sql 驱动注册到了 sql 包中。这样只要应用层代码在 Open 数据库的时候，传入驱动的名字（这里是“postgres”），那么通过 sql.Open 函数，返回的数据库实例句柄对数据库进行的操作，实际上调用的都是 pq 包中相应的驱动实现。

实际上，这种**通过在 init 函数中注册自己的实现的模式，就有效降低了 Go 包对外的直接暴露**，尤其是包级变量的暴露，从而避免了外部通过包级变量对包状态的改动。

另外，从标准库 database/sql 包的角度来看，这种“注册模式”实质是一种工厂设计模式的实现，sql.Open 函数就是这个模式中的工厂方法，它根据外部传入的驱动名称“生

产”出不同类别的数据库实例句柄。

这种“注册模式”在标准库的其他包中也有广泛应用，比如说，使用标准库 image 包获取各种格式图片的宽和高：

复制代码

```
package main

import (
    "fmt"
    "image"
    _ "image/gif" // 以空导入方式注入gif图片格式驱动
    _ "image/jpeg" // 以空导入方式注入jpeg图片格式驱动
    _ "image/png" // 以空导入方式注入png图片格式驱动
    "os"
)

func main() {
    // 支持png, jpeg, gif
    width, height, err := imageSize(os.Args[1]) // 获取传入的图片文件的宽与高
    if err != nil {
        fmt.Println("get image size error:", err)
        return
    }
    fmt.Printf("image size: [%d, %d]\n", width, height)
}

func imageSize(imageFile string) (int, int, error) {
    f, _ := os.Open(imageFile) // 打开图文文件
    defer f.Close()

    img, _, err := image.Decode(f) // 对文件进行解码，得到图片实例
    if err != nil {
        return 0, 0, err
    }

    b := img.Bounds() // 返回图片区域
    return b.Max.X, b.Max.Y, nil
}
```

你可以看到，上面这个示例程序支持 png、jpeg、gif 三种格式的图片，而达成这一目标的原因，正是 image/png、image/jpeg 和 image/gif 包都在各自的 init 函数中，将自己“注册”到 image 的支持格式列表中了，你可以看看下面这个代码：

```
// $GOROOT/src/image/png/reader.go
func init() {
    image.RegisterFormat("png", pngHeader, Decode, DecodeConfig)
}

// $GOROOT/src/image/jpeg/reader.go
func init() {
    image.RegisterFormat("jpeg", "\xff\xd8", Decode, DecodeConfig)
}

// $GOROOT/src/image/gif/reader.go
func init() {
    image.RegisterFormat("gif", "GIF8?a", Decode, DecodeConfig)
}
```

复制代码

那么，现在我们了解了 init 函数的常见用途。init 函数之所以可以胜任这些工作，恰恰是因为它在 Go 应用初始化次序中的特殊“位次”，也就是 main 函数之前，常量和变量初始化之后。

小结

好了，我们今天这一节课就到这里了。

在这一节课中，我们一起了解了 Go 应用的用户层入口函数 main.main、包初始化函数 init，还有 Go 程序包的初始化次序和包内各种语法元素的初始化次序。

其中，你需要重点关注 init 函数具备的几种行为特征：

> 执行顺位排在包内其他语法元素的后面；
>
> 每个 init 函数在整个 Go 程序生命周期内仅会被执行一次；
>
> init 函数是顺序执行的，只有当一个 init 函数执行完毕后，才会去执行下一个 init 函数。

基于上面这些特征，init 函数十分适合做一些包级数据初始化工作以及包级数据初始状态的检查工作，我们也通过实例讲解了 init 函数的这些常见用途。

最后，大多 Go 程序都是并发程序，程序会启动多个 Goroutine 并发执行程序逻辑，这里你一定要注意主 Goroutine 的优雅退出，也就是主 Goroutine 要根据实际情况来决定，

是否要等待其他子 Goroutine 做完清理收尾工作退出后再行退出。

思考题

今天我给你留了一个思考题：当 init 函数在检查包数据初始状态时遇到失败或错误的情况，我们该如何处理呢？欢迎在留言区留下你的答案。

感谢你和我一起学习，也欢迎你把这门课分享给更多对 Go 语言感兴趣的朋友。我是 Tony Bai，我们下节课见。

分享给需要的人，Ta订阅后你可得 20 元现金奖励

生成海报并分享

赞 1 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 07｜构建模式：Go Module的6类常规操作

精选留言 (10)

写留言

2021-10-29

init一般都是初始化一些东西，如果失败，可能会影响实际的运行，因此我们目前的操作，就是记录Error日志，然后os.Exit(-1)。
另外麻烦老师看一下我第6讲留言的问题，谢谢老师

1

liaomars

2021-10-29

老师，在演示包的执行顺序的示例中25行，c1 应该是c2 吧，
if c1 != "" { fmt.Println("main: const c2 has been initialized") }

1 1

二的根比方
2021-10-29

老师，请问下“GO入口文件包名称一定要是main，func名称一定要是main，（满足2个条件）文件名或者文件夹名字不一定是main。”这样的说法对吗

展开

jc9090kkk
2021-10-29

一般是根据初始化严重程度的级别来抛出不同的错误，若是较为严重的影响程序生命周期的错误，则直接抛出运行时恐慌(Panic)，如果是较低级别的错误，可以通过记录trance等提供相应错误日志

2021-10-29

我再导入自己写的pkg1的包的时候 导入不进去，本地的pkg包要用怎么办。

歪文
2021-10-29

当 init 函数在检查包数据初始状态时遇到失败或错误的情况，我们该如何处理呢
1. 由于是程序启动，可以直接panic，让开发者解决问题。
2. 如果不影响整个程序运行，打印warn警告，然后使用默认值代替，继续运行下去。

展开

liaomars
2021-10-29

Q：当 init 函数在检查包数据初始状态时遇到失败或错误的情况，我们该如何处理呢
A：目前是Go小白，因为在init里面执行的是一些初始化的操作，为后面的代码做铺垫用的，现在初始化错误或失败那肯定会影响后面的代码运行，所以我觉得应该是中止代码的执行。

展开

andox
2021-10-29

分情况而定
1. 初始化失败的是必要的数据 panic处理 结束进程

1. 初始化失败的是对业务没影响，可成功可失败的 输出warn或error日志 方便定位

我爱北京天安门

2021-10-29

一口气看完落下的三篇，很好的课程，期待后面

展开

Calvin

2021-10-29

简单做个笔记：

- Go 包的初始化次序：

1）依赖包按“深度优先”的次序进行初始化；

2）每个包内按以“常量 -> 变量 -> init 函数”（main.main 函数前）的顺序进行初始化；

3）包内的多个 init 函数按出现次序进行自动调用。...

展开